

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Espacio libre y paginación

De huecos de tamaño variable a páginas de tamaño fijo

Gerson Garrido

Semana 6

Hoy

Sesión A · Gestión de espacio libre

1. La lista libre: división y fusión
2. El encabezado: por qué `free` no pide el tamaño
3. Best, worst, first y next fit
4. Listas segregadas, slab y buddy
5. `kalloc` en xv6

Sesión B · Paginación

6. Páginas, marcos, VPN y offset
7. La tabla de páginas y sus entradas
8. Traducir a mano, bit por bit
9. Costo en espacio y en tiempo
10. Sv39: la PTE de xv6-riscv

> **Lectura**
>
> OSTEP cap. 17 *Free-Space Management* y cap. 18 *Paging: Introduction*.

SECCIÓN 01

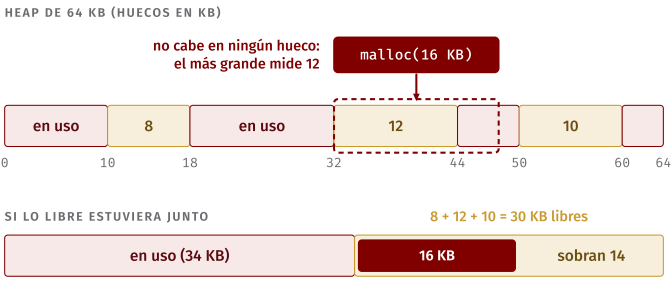
La lista libre

01



El problema: pedazos de tamaño variable

`malloc`/`free` y la segmentación reparten pedazos de tamaño variable. Con el tiempo, lo libre queda partido en huecos: hay espacio en total, pero ningún hueco alcanza.

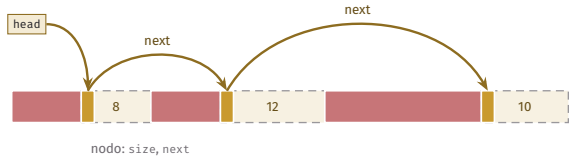


Pero malloc no puede mover los bloques en uso: el programa guarda punteros a ellos.

Supuestos de hoy (como OSTEP): interfaz `malloc`/`free`; un bloque entregado no se mueve, así que no se puede compactar.

La lista libre *(free list)*

- Una lista enlazada con un nodo por **hueco**: dónde empieza y cuánto mide.
- ¿Dónde se guardan los nodos? No hay otro `malloc` al que pedirle memoria…
- …así que el nodo vive dentro del propio hueco: sus primeros bytes guardan `size` y `next`.

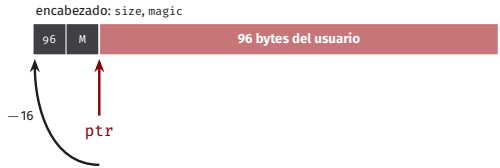


mymalloc.c

```
1  typedef struct node {
2      size_t size;        // bytes libres
3      struct node *next;  // otro hueco
4  } node_t;
```

El encabezado: por qué `free` no pide el tamaño

- `free(ptr)` recibe solo un puntero. ¿Cómo sabe cuántos bytes liberar?
- `malloc` guarda un **encabezado** *(header)* justo antes del bloque que entrega: el tamaño y un número mágico.
- `free` retrocede 16 bytes y lo lee. `malloc(n)` consume $n + 16$ bytes (y redondea n a 16).
- El número mágico detecta un `free` de un puntero que no vino de `malloc`.

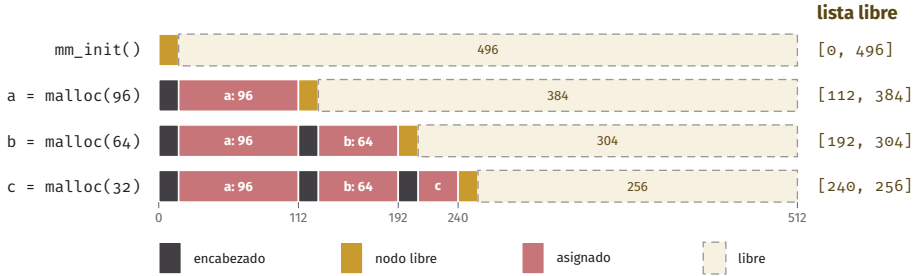


mymalloc.c · mm_free()

```
1  typedef struct header {
2      size_t size;   // bytes útiles
3      size_t magic;  // MAGIC
4  } header_t;
5
6  header_t *h = (header_t *)ptr - 1;
7  if (h->magic != MAGIC) abort();
```

División (*splitting*): tres `malloc` sobre 512 bytes

`mm_init()` deja un solo trozo libre: un nodo de 16 bytes y 496 libres. Cada `malloc` toma el comienzo del trozo y divide el resto en un trozo libre nuevo.



Cada bloque asignado ocupa su tamaño +16: `a` empieza en el byte 16 y el siguiente trozo, en el 112.

Fusión *(coalescing)*: tres `free`

Al liberar, el bloque vuelve a la lista ordenada por dirección. Si el trozo vecino también está libre, se fusionan en uno solo.



Sin fusión, los mismos tres `free`:



- `free(a)`: `a` termina donde empieza el trozo de `b`. Se suman los tamaños *y* el nodo que sobra (16 bytes).
- Sin fusión hay 448 bytes libres, pero `malloc(300)` devuelve NULL: fragmentación externa fabricada por el propio asignador.

El código: dividir al asignar

code/week-06/mymalloc.c · mm_malloc()

```
1   node_t **link = &head;
2   for (node_t *cur = head; cur != NULL;
3        link = &cur->next, cur = cur->next) {
4     if (cur->size < size)
5       continue;                       // first fit
6     if (cur->size >= size + sizeof(node_t) + ALIGN) {
7       node_t *rest = (node_t *)((char *)cur
8                       + sizeof(header_t) + size);
9       rest->size = cur->size - size - sizeof(node_t);
10      rest->next = cur->next;
11      *link = rest;                   // división
12    } else {
13      size = cur->size;               // completo
14      *link = cur->next;
15    }
16    // ... escribir el encabezado y retornar h + 1
```

- `link` apunta al `next` anterior: así se reemplaza un nodo sin casos especiales para la cabeza.
- Si sobra muy poco, no se divide: un trozo de 0 bytes no sirve. Esos bytes de más son fragmentación **interna**.
- El nodo libre se convierte en encabezado: mismo lugar, mismo tamaño (16 bytes).

El código: fusionar al liberar

code/week-06/mymalloc.c · mm_free()

```c
node_t *prev = NULL, *cur = head;
while (cur != NULL && cur < n) {  // orden
  prev = cur; cur = cur->next;
}
n->next = cur;
if (prev == NULL) head = n; else prev->next = n;
if (cur != NULL && adjacent(n, cur)) {
  n->size += sizeof(node_t) + cur->size;
  n->next = cur->next;
}
if (prev != NULL && adjacent(prev, n)) {
  prev->size += sizeof(node_t) + n->size;
  prev->next = n->next;
}
```

- `adjacent(a, b)`: ¿`a` acaba donde empieza `b`?
- Fusiona con el vecino derecho y luego con el izquierdo.
- Costo: $O(n)$ para mantener el orden.
- `heaptest` libera tres bloques en los 6 órdenes.

bash

```
$ make test
./heaptest
heaptest: todas las pruebas pasaron
```

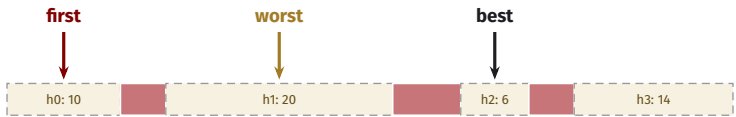
SECCIÓN 02

Estrategias de asignación

02

¿Qué hueco elegir?

Huecos de 10, 20, 6 y 14 KB, en orden de dirección. Llega `malloc(4)`:



- **First fit**: el primer hueco que alcance. Rápido: se detiene apenas encuentra uno.
- **Best fit**: el hueco más chico que alcance. Recorre toda la lista; deja sobras pequeñas.
- **Worst fit**: el hueco más grande, para que la sobra siga siendo útil. También recorre todo.
- **Next fit**: como first fit, pero busca desde donde se quedó la última vez.

Empates: gana el hueco de menor dirección. Modelo simple, como `malloc.py`: sin encabezados ni fusión.

La misma secuencia, cuatro resultados



- **Worst** gasta el hueco grande en peticiones chicas; **next** se separa de first en la tercera petición: no vuelve a `h0`, sigue en `h1`.
- Aquí gana best fit, pero hay secuencias donde pierde: deja astillas (2 KB) que nadie usa.

No hay una estrategia ganadora

code/week-06

```
$ ./fits 10,20,6,14 4,8,4,16
huecos iniciales [10 20  6 14]

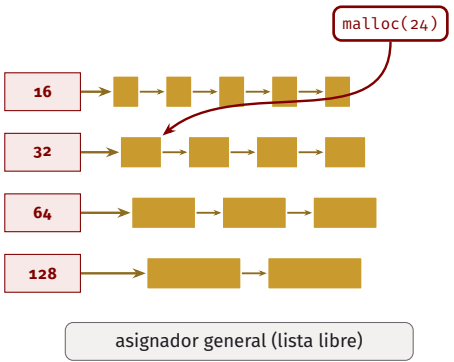
política    4   8   4  16   huecos al final
best       h2  h0  h3  h1   [ 2  4  2 10]
worst      h1  h1  h3  --   [10  8  6 10]  1 sin hueco
first      h0  h1  h0  --   [ 2 12  6 14]  1 sin hueco
next       h0  h1  h1  --   [ 6  8  6 14]  1 sin hueco
```

Prueben otras secuencias: es fácil construir una donde best fit falla y worst no.

- **Costo**: best y worst recorren *toda* la lista; first y next, solo hasta el primero.
- First fit acumula astillas al inicio de la lista; next fit reparte el desgaste.
- Con la lista ordenada por dirección, first fit además facilita la fusión.
- En el laboratorio: `malloc.py -p BEST`, `WORST`, `FIRST`, con y sin `-C`.

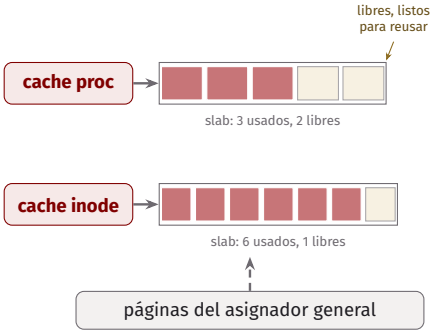
Listas segregadas *(segregated lists)*

- Muchos programas piden una y otra vez los mismos tamaños.
- Idea: una lista aparte por **clase de tamaño**. Pedir 24 bytes es sacar el primero de la lista de 32: $O(1)$, sin buscar ni dividir.
- Lo que no cae en ninguna clase va al asignador general.
- Problema: ¿cuánta memoria le toca a cada lista? Si la de 64 se vacía mientras la de 16 sobra, hay que mover memoria entre ellas.



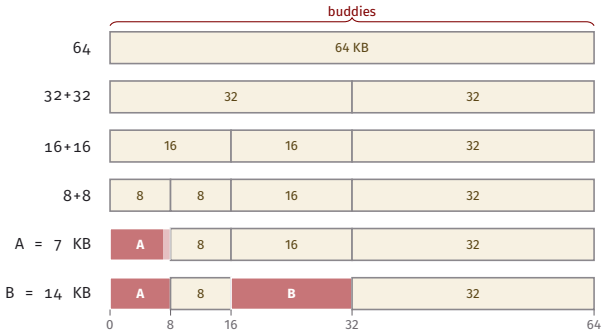
Slab allocator

- Bonwick, Solaris (1994). Hoy también en Linux.
- Un **cache** por tipo de objeto del kernel: `proc`, `inode`, locks…
- Cada cache tiene **slabs**: unas páginas partidas en objetos de ese tamaño. Si un cache se queda sin espacio, pide otro slab al asignador general.
- Los objetos libres se guardan ya inicializados: crear y destruir objetos cuesta menos que armarlos desde cero.



Buddy allocator: dividir en mitades

- Todo bloque mide una **potencia de 2**. Arena de 64 KB.
- Pedido de 7 KB: se divide a la mitad hasta llegar al bloque más chico que alcance, 8 KB. Las dos mitades de una división son **buddies**.
- Se entregan 8 KB por 7 pedidos: 1 KB de fragmentación interna.
- Luego, 14 KB: el bloque libre de 16 KB alcanza sin dividir.

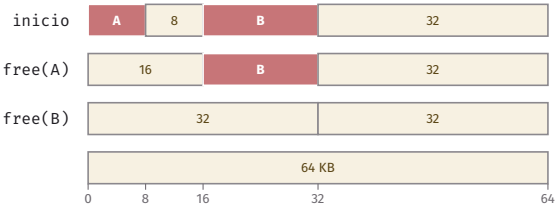


Buddy allocator: liberar y fusionar

- Al liberar un bloque, se mira a su **buddy**. Si está libre, se fusionan y se repite un nivel más arriba.
- Encontrar al buddy es barato. Un bloque de tamaño s en la dirección a tiene a su buddy en

$$a \oplus s \quad \text{(XOR)}$$

  Las direcciones de los dos difieren en un solo bit.
- Fusión recursiva y rápida. A cambio, fragmentación interna: pedir 33 KB cuesta 64.



Bloque	**Buddy** ($a \oplus s$)	**¿Libre?**
A: $a = 0, s = 8$	$000000 \oplus 001000 = 8$	sí: fusionar
$a = 0, s = 16$	$000000 \oplus 010000 = 16$	no (B): parar
B: $a = 16, s = 16$	$010000 \oplus 010000 = 0$	sí: fusionar
$a = 0, s = 32$	$000000 \oplus 100000 = 32$	sí: fusionar

En xv6: `kalloc` y `kfree`

kernel/kalloc.c

```
1  struct run {
2    struct run *next;
3  };
4  void
5  kfree(void *pa)
6  {
7    struct run *r;
8    // ... (verifica alineación y rango)
9    memset(pa, 1, PGSIZE);  // basura
10   r = (struct run *)pa;
11   acquire(&kmem.lock);
12   r->next = kmem.freelist;
13   kmem.freelist = r;
14   release(&kmem.lock);
15 }
```

`kalloc()` hace lo inverso: saca la cabeza.

- Una lista libre real: el nodo (`struct run`) vive dentro de la página libre.
- Todos los bloques miden **4096 bytes**. No hay división, ni fusión, ni estrategia: cualquier página sirve. Por eso no hay fragmentación externa.
- Es una pila (LIFO): `kfree` empuja, `kalloc` saca.



`kinit()` libera de `end` a `PHYSTOP`: la última liberada, `PHYSTOP` — 4096 = `0x87fff000`, queda primera.

SECCIÓN 03

Paginación

03

La idea: partir todo en pedazos iguales

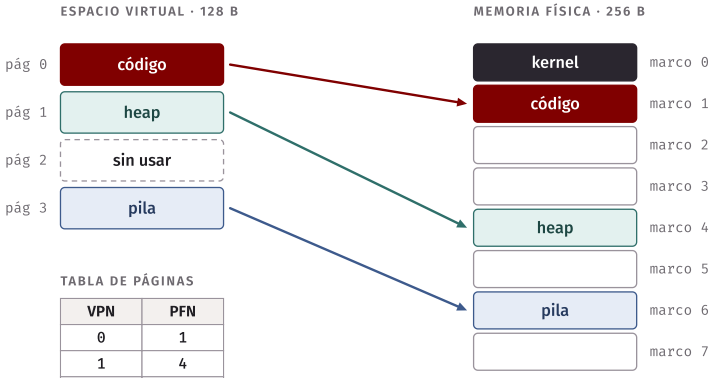
- Lo variable trae fragmentación externa. Solución radical: todo del mismo tamaño.
- El espacio de direcciones se parte en **páginas** *(pages)*; la memoria física, en **marcos de página** *(page frames)* del mismo tamaño.
- Cualquier página puede ir en cualquier marco libre. Las páginas no tienen que estar contiguas ni en orden.
- Para cada proceso, una **tabla de páginas** *(page table)* dice en qué marco está cada página.



Atlas, Universidad de Manchester, 1963. Iain MacCallum, CC BY 3.0, Wikimedia Commons.

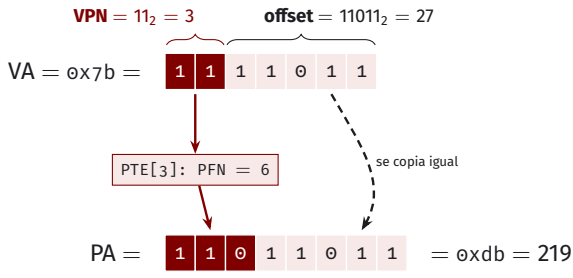
La Atlas (1962) fue la primera computadora con paginación.

Páginas y marcos: un ejemplo de juguete



VPN y offset

Páginas de $32 = 2^5$ bytes: los 5 bits bajos dicen dónde dentro de la página (el **offset**). Los bits restantes son el **número de página virtual** *(virtual page number, VPN)*.



- Solo se traduce el VPN. El offset pasa tal cual: una página entera va a un marco entero.
- Con shifts y máscaras: `vpn = va >> 5`, `offset = va & 0x1f`, `pa = (pfn << 5) | offset`.

Qué hay en una entrada de la tabla de páginas

Una PTE *(page table entry)* guarda el número de marco *(physical frame number, PFN)* y bits de control. Ejemplo genérico de 32 bits:



- **Válido** *(valid)*: ¿la página existe en el espacio de direcciones? Las páginas sin usar (entre heap y pila) quedan inválidas y no gastan marco.
- **Protección** (R, W, X, U): ¿se puede leer, escribir, ejecutar? ¿Desde modo usuario?
- **Presente** *(present)*: ¿está en memoria o se envió a disco (swap)? Semanas 9 y 10.
- **Sucio** *(dirty)*: ¿se escribió desde que se cargó? Si se reemplaza, hay que guardarla.
- **Referencia** *(accessed)*: ¿se usó hace poco? Ayuda a elegir qué página reemplazar.

Traducir `0x3229` paso a paso (páginas de 1 KB)



TABLA DE PÁGINAS

VPN	PFN
0	5
1	12
2	—
3	—
4	30
5	—
6	—
7	—
8	—
9	—
10	—
11	—
12	29
13	—
14	—
15	7

PTE[12] válida: PFN = 29 = 11101

PA = 0x7629 comprobación: 29 × 1024 + 553 = 30249 = 0x7629

Más direcciones, con el calculador

code/week-06

```
$ ./translate 1024 \
    5,12,-,-,30,-,-,-,-,-,-,-,29,-,-,7 \
    0x3229 0x13ff 0x0a0f
...
VA 0x13ff = 0100 | 1111111111
  VPN = 4, offset = 0x3ff
  PTE[4]: PFN = 30 = 0x1e
  PA = 11110 | 1111111111 = 0x7bff

VA 0x0a0f = 0010 | 1000001111
  VPN = 2, offset = 0x20f
  PTE[2] inválida -> excepción
```

- `0x13ff`: el último byte de la página 4 cae en el último byte del marco 30.
- `0x0a0f`: la página 2 no está en uso. La MMU no traduce: lanza una **excepción** y el kernel decide. Normalmente, *segmentation fault*.
- Antes de correrlo, háganlo a mano: es la pregunta de repaso 2.

Salida recortada: sin el encabezado, sin `0x3229` (diapositiva anterior) y sin el final de la última línea.

Lo que hace la MMU en cada acceso

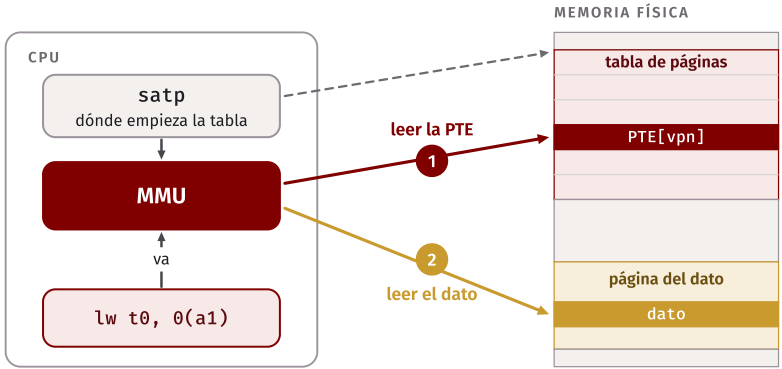
pseudocódigo

```
vpn    = va >> OFFSET_BITS;
offset = va & (PAGE_SIZE - 1);
// la tabla está en memoria: ¡un acceso más!
pte = mem[ptbr + vpn * sizeof(pte)];
if (!pte.valid)
  trap(PAGE_FAULT);       // decide el kernel
if (!permitted(pte, kind_of_access))
  trap(PROTECTION_FAULT);
pa  = (pte.pfn << OFFSET_BITS) | offset;
val = mem[pa];            // el acceso pedido
```

- La tabla de páginas es demasiado grande para la MMU: vive en memoria física, en páginas del kernel.
- Un registro, el **page table base register** (PTBR), dice dónde empieza la tabla del proceso actual. En RISC-V se llama `satp`.
- En cada cambio de contexto, el kernel cambia ese registro: cada proceso tiene su propia tabla.

El costo en tiempo: cada acceso del programa (instrucción o dato) lee **antes** su PTE. El doble de accesos a memoria.

La tabla vive en memoria: dos accesos por cada uno

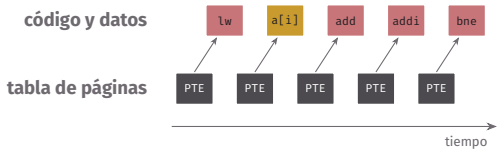


un acceso del programa = dos accesos a memoria

Cuántos accesos cuesta un bucle

el bucle y su RISC-V

```
1  for (i = 0; i < 1000; i++)
2    sum += a[i];
3
4  loop: lw   t0, 0(a1)
5        add  a0, a0, t0
6        addi a1, a1, 4
7        bne  a1, a2, loop
```



la mitad de los accesos solo sirve para traducir

- Por vuelta: 4 instrucciones que leer y 1 dato (`lw`). 5 accesos.
- Con paginación: 10. Y 10 000 en todo el bucle, en vez de 5000.

Próxima semana: esas PTEs se repiten; la **TLB**, una caché de traducciones, evita casi todos esos accesos.

El costo en espacio: ¿cuánto mide la tabla?

32 bits, páginas de 4 KB, PTE de 4 bytes

- Offset: 4 KB $= 2^{12} \Rightarrow$ 12 bits.
- VPN: $32 - 12 = 20$ bits $\Rightarrow 2^{20} \approx$ un millón de PTEs.
- Tamaño: $2^{20} \times 4 = 2^{22}$ bytes $=$ 4 MB por proceso.
- Con 100 procesos: 400 MB solo en tablas, casi todas las PTEs inválidas.

64 bits, páginas de 4 KB, PTE de 8 bytes

- x86-64 usa 48 bits: 2^{36} PTEs $\times 8 = 2^{39} =$ 512 GB.
- Sv39 (xv6): 39 bits: $2^{27} \times 8 = 2^{30} =$ 1 GB.
- Los 64 bits completos: $2^{52} \times 8 = 2^{55} =$ 32 PB.

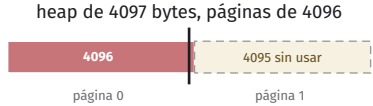
Verifíquenlo: `./ptsize 48 4096 8`.

Una tabla lineal no sirve en 64 bits

Por eso las tablas reales son **multinivel**: solo existen las partes que se usan. Semana 7, y el proyecto P3.

¿Qué fragmentación tiene la paginación?

- **Externa: ninguna**. Todo marco libre sirve para cualquier página: no hay huecos “demasiado chicos”.
- **Interna: sí**. La memoria se entrega por páginas enteras. Un heap de 4097 bytes ocupa 2 páginas y desperdicia 4095.
- En promedio, media página por región (código, heap, pila…).
- Y el espacio de las propias tablas: 4 MB por proceso en 32 bits.



Pregunta de repaso 3

Paginación: sin fragmentación externa, con interna (la última página de cada región).
Segmentación: al revés.

SECCIÓN 04

Paginación en xv6-riscv

04



Direcciones Sv39

Modo **Sv39**: direcciones de 39 bits, páginas de 4 KB, tres niveles de tablas de 512 PTEs de 8 bytes.



kernel/riscv.h

```
1  #define PXMASK          0x1FF // 9 bits
2  #define PXSHIFT(level)  (PGSHIFT + (9 * (level)))
3  #define PX(level, va)   ((((uint64)(va)) >> PXSHIFT(level)) & PXMASK)
```

./sv39

```
0x40201abc  PX(2,1,0) =   1   1   1  offset 0xabc
TRAMPOLINE  PX(2,1,0) = 255 511 511  offset 0x000
```

`PX(2)` de `TRAMPOLINE` da 255, no 511: xv6 usa un bit menos (`MAXVA` $= 2^{38}$) para evitar la extensión de signo.

La PTE de Sv39



- **V**: válida. RISC-V no tiene un bit *presente* aparte: V cumple los dos papeles.
- **R, W, X**: leer, escribir, ejecutar. **U**: accesible en modo usuario.
- Si V = 1 y R = W = X = 0, la PTE no es una hoja: apunta al siguiente nivel. `walk()` en `kernel/vm.c` cuenta con eso.
- **A** (referencia) y **D** (sucio) existen en el hardware, pero `riscv.h` no los define: xv6 no hace swap.

kernel/riscv.h

```
1  #define PTE_V (1L << 0) // valid
2  #define PTE_R (1L << 1)
3  #define PTE_W (1L << 2)
4  #define PTE_X (1L << 3)
5  #define PTE_U (1L << 4) // user can access
```

PA2PTE y PTE2PA con números

kernel/riscv.h

```
1  // shift a physical address to the right place for a PTE.
2  #define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)
3  #define PTE2PA(pte) (((pte) >> 10) << 12)
4  #define PTE_FLAGS(pte) ((pte) & 0x3FF)
```

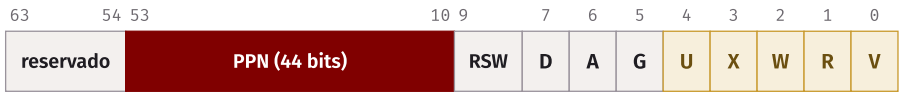
./sv39

```
pa          = 0x87f42000
PA2PTE(pa)  = 0x21fd0800
pte         = 0x21fd0817  U-WRV
PTE_FLAGS   = 0x017
PTE2PA(pte) = 0x87f42000
```

- » `12` quita el offset: queda el PPN `0x87f42`.
- « `10` lo corre por encima de los 10 bits de flags.
- | `0x17` agrega U, W, R y V. Así arma cada PTE `mappages()` en `kernel/vm.c`: `PA2PTE(pa) | perm | PTE_V`.
- `PTE2PA` hace lo inverso: borra los flags y devuelve la dirección del marco.

La PTE `0x21fd0817`, campo por campo

UNA PTE DE SV39: 64 BITS



EJEMPLO DE SV39.C: `pte = 0x21fd0817`



PTE2PA: `0x87f42 << 12 = 0x87f42000`

U, W, R, V: página de usuario, lectura y escritura (0x17)

V válida · R lectura · W escritura · X ejecución · U accesible en modo usuario

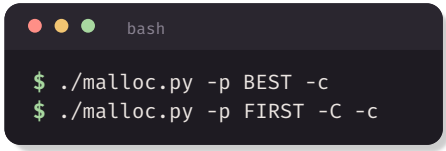
G global · A accedida · D sucia (en gris: xv6 no las usa) · V = 1 con R = W = X = 0: apunta al siguiente nivel

Resumen

- Un asignador de tamaño variable lleva una **lista libre** dentro de los propios huecos; **divide** al asignar y **fusiona** al liberar. El **encabezado** guarda el tamaño para `free`.
- Best, worst, first y next fit: ninguna gana siempre. Todas sufren fragmentación externa.
- Listas segregadas y slab: rápidos para tamaños frecuentes. Buddy: fusión barata con XOR, a cambio de fragmentación interna. `kalloc` de xv6: páginas de un solo tamaño, una pila.
- **Paginación**: VA $=$ VPN $|$ offset; la tabla da el PFN; PA $=$ PFN $|$ offset. Sin fragmentación externa.
- Costos: una tabla lineal de 4 MB por proceso en 32 bits (imposible en 64) y un acceso extra a memoria por cada referencia.
- Sv39: PTE de 64 bits con PPN y flags V, R, W, X, U; `PA2PTE`, `PTE2PA` y `PX` en `kernel/riscv.h`.

Para la próxima clase

- Leer OSTEP cap. 19 *TLBs* y cap. 20 *Smaller Tables*.
- **Laboratorio**: simuladores de OSTEP.
  - `relocation.py`: preguntas 1–5 del cap. 15.
  - `segmentation.py`: preguntas 1–5 del cap. 16.
  - Tarea de código del cap. 14 con `gdb` y `valgrind`.
  - `malloc.py`: `-p BEST`, `WORST`, `FIRST`, con y sin `-C`.
- **P1** (`wish`): entrega esta semana.
- **P2**: entrega en la semana 8. Parcial: semana 8, sesión B.



Pregunta para pensar

Si cada acceso a memoria cuesta el doble con paginación, ¿por qué los programas no corren a la mitad de velocidad?